

**Design and Analysis of Algorithms, Chennai Mathematical Institute**  
**Prof. Madhavan Mukund**  
**Department of Computer Science and Engineering,**

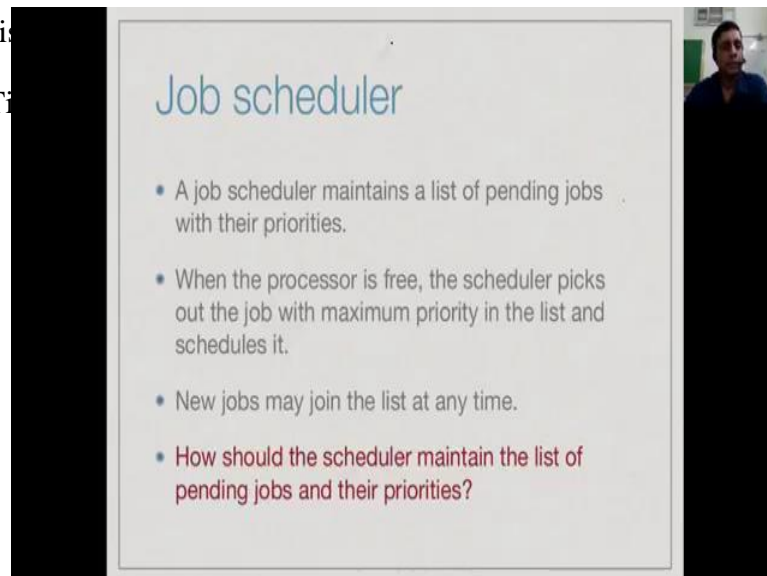
**Week - 05**  
**Module - 03**  
**Lecture - 34**  
**Priority queues**

So, we have seen how the use of a clever data structure, the union find data structure can make Kruskal's algorithm more efficient. The other two algorithms which we need to

make efficient are Dijkstra's algorithm for the single source shortest path, and Prim's algorithm for the minimum spanning tree. Both of them it turns out required a data

structure that is

(Refer Slide T



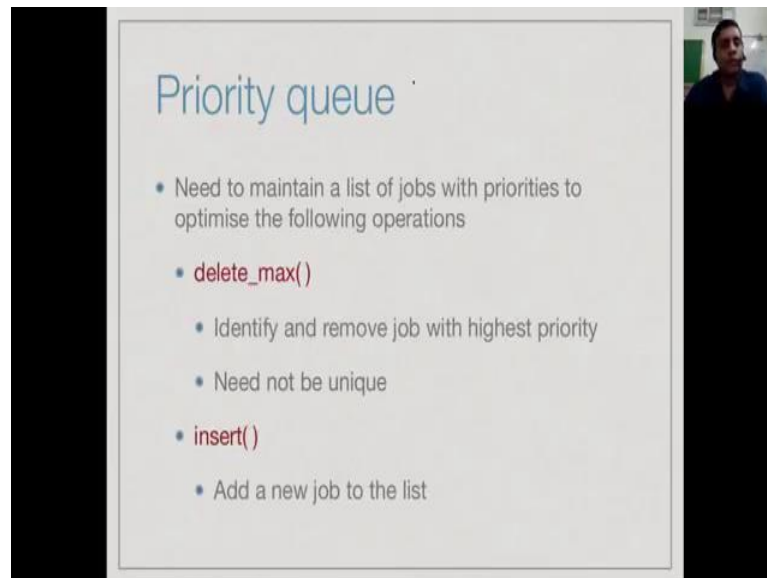
**Job scheduler**

- A job scheduler maintains a list of pending jobs with their priorities.
- When the processor is free, the scheduler picks out the job with maximum priority in the list and schedules it.
- New jobs may join the list at any time.
- How should the scheduler maintain the list of pending jobs and their priorities?

So, to understand priority queues, let us look at the following scenario. So, supposing we would have a job scheduler running on an operating system. So, when we run multiple tasks on an operating system each of them runs for a little bit of time, and then it goes out. So, a job scheduler maintains a list of all jobs which are pending along with their priorities. Whenever the processor is free, the scheduler picks out a job with a maximum priority in the list and schedules it. Now what happens? Of course, this is dynamic. So, while some jobs are running, other new jobs may arrive. So, each time when new jobs arrive they will come with different priorities, and this is the job of the scheduler to make sure that the new jobs with higher priority come ahead of older jobs with lower priority.

So, from a data structure point of view, our question is what is the best way for the scheduler to maintain the list of pending jobs with their priorities? So, that this list can be updated dynamically as this list gets updated ((Refer Time: 01:22)) new jobs arrive from.

(Refer Slide Time: 01:26)



The slide is titled "Priority queue" in a blue font. It contains a bulleted list of operations and their descriptions. A small inset video of a person is visible in the top right corner of the slide area.

- Need to maintain a list of jobs with priorities to optimise the following operations
  - `delete_max()`
    - Identify and remove job with highest priority
    - Need not be unique
  - `insert()`
    - Add a new job to the list

So, this is what we call a priority queue? So, we have two basic operations in a priority queue; the first is to extract the next job which in this case means take the job with the highest priority, and remove it from the queue. Now of course, we have the situation where you have multiple jobs in the same priority. So, we do not assume the priorities are you need, but if they are not you need then we can assume either the there is some type writing rule or we do not care which one of the highest priority jobs we get right. So, the main operation is delete max; delete the maximum priority item from the queue. And then there is an insert. So, a job will come it will be inserted, it will come with a priority. So, we need to insert into the list. So, that subsequent delete max take this one in to a count.

(Refer Slide Time: 02:09)

Linear structures

- Unsorted list
  - `insert()` takes  $O(1)$  time
  - `delete_max()` takes  $O(n)$  time
- Sorted list
  - `delete_max()` takes  $O(1)$  time
  - `insert()` takes  $O(n)$  time
- Processing a sequence of  $n$  jobs requires  $O(n^2)$  time

So, the first solution that one could think of to keep such a data is to maintain a list, some kind of a linear structure and array or a list, since it is growing dynamically a list is your obvious choice. Now if we keep an unsorted list, then it is easy to add a job, which is up to appended to the list. So, insertion is a constant time operation takes time  $O(1)$ , but as we have seen many times if we have an unsorted list, then we have to scan the entire list to find the minimum or the maximum. So, in this case if we want to do a delete max from an unsorted list, this operation is going to take as linear time proportional to the size of the list or the number of jobs in a pending at the moment.

So, the only useful thing one can do with the list, otherwise it should sort it. So, if we sorted we can try and maintain jobs in decreasing order of priority, if we do it decreasing order of priority, then the maximum priority job is always at the beginning of the list. So, we can instantly find it in constant time. So, delete max now becomes a constant time operation, then what happens to insert? We need to insert a value in to the list, and I resolve it insertion sort when we insert a value in to the sorted list we have to walked on the list to find the correct list to put it in. So, that will now take order  $N$  time. So, between an unsorted list or in a sorted list, we have a trade off in an unsorted list delete max takes linear time, in a sorted list insert takes linear time. So, one all the other is a bottle neck.

So, whether we use an unsorted list or a sorted list over a sequence of  $N$  jobs, supposing  $N$  jobs arrive in the system will be process them. Then we will be inserting them  $N$  times and deleting max  $N$  times. So, which ever solution we use among these two, we will end

up spending order  $N$  square time. So, this suggest to us that a linear structure. One dimensional structure has a severe limitation with respect to solving the representation of the priority queue in an efficient way.

(Refer Slide Time: 04:13)

**Two dimensional structures**

$N = 25$

**First attempt**

- Assume  $N$  processes enter/leave the scheduler
- Keep an  $\sqrt{N} \times \sqrt{N}$  array
- Each row is maintained in sorted order

12	17	29	31	40
8	19	22	33	37
10	13	14		
13	20	25	43	
6	11			

So, we have to go from a one dimensional structure to a two dimensional structure. So, let us start with a very naive two dimensional structure, just to show how we can get drastic improvements, just I moving from one dimension, two dimensions. So, here we assume that we know an advance somehow about in a number of total jobs we can have. So, we have assumed in this case say that  $N$  is 25. So, now what we do is instead of maintaining a one dimensional list of length 25 - maximum length 25, we reorganize this list as a squared array of 5 by 5. So, square root  $N$  by square root  $n$ . So, now you can see here an example of such a thing. So, we have 25 minus 6 19 jobs currently with different priorities in there. Sometimes two jobs may have the same priority that does not matter. But the important thing is that we are not necessarily maintaining it in a kind of sequential way through this two dimensional array, all we are guarantying is that every row as we look at the row from left to right is an ascending order.

So, we have the first row is an ascending order, second row in, but the rows themselves have nothing in common ((Refer Time: 05:22)). So, you could have bigger values in the second row in the smaller row and vice versa. So, for example, 19 is bigger than 12, then the 17 is bigger than 8 and so on. So, these are kind of five independent list of jobs,



which together makeup the total set of jobs at each list is sorted.

(Refer Slide Time: 05:40)

insert()

Insert 11

- Insert into first row that has free space
- Maintain size of each row
- Takes time  $\sqrt{N}$

12	17	29	31	40	5
8	19	22	33	37	5
10	11	13	14		3
13	20	25	43		4
6	11				2

So, now suppose you want to insert a new job in to this list. So, strategy is very simple we want to insert it in the correct place in the first row that has free space. Now to find out if one of the rows has free space, we need to walk down this thing to find out how many elements are there. So, you will save that work by keeping this extra information here, which is the size of each of the rows. So, let us assume that we have the size available of insert the rows, now we row in this particular case that each row is a size five at most. So, if the size is 5, when we know that if it try to insert 11 into this first row, the 11 will not be able to fit there, because there is no space basically. So, then we move to the next row, once again we see that the size is 5. So, then we go to the third. Now we see that there is a space, because there are only three elements in this row and it can take 5. So, now we work down this list and find the correct place to insert it, and insert it as we doing insertion sort.

So, another question is how much time this is take. Well, we know that it we have square root of  $N$  rows. So, it take square root of  $N$  steps to find the correct row to insert going from top to bottom, and then we have in each rows square root of  $N$  entries. So, if we do a usual insertion it takes time proportional to the length of the row, we are inserting into. So, we have a square root of  $N$  scan to find the correct row, the square root of  $N$  scan to find the correct position within that row. So, overall it is time  $O$  order of square root of  $n$ . So, what about deleting?

(Refer Slide Time: 07:15)

`delete_max()`

- Maximum in each row is the last element
- Maximum among these is to be deleted
- Again  $O(\sqrt{N})$

12	17	29	31	40	5
8	19	22	33	37	5
10	11	13	14		4
13	20	25			3
6	11				2

So, we want to delete the maximum element in this array. Now because each row is independent of other row, we do not know in advance where this maximum element is. However, we do know that in every row the maximum element is at the end, because each row is sorted. So, the first thing we do know is that every possible maximum element is actually at the end of its row. So, we have these five rows, we have five maximum elements. And now the overall maximum must be among these five elements, it must be the largest one. So, we can just use the size information. So, we know it is the fifth element in the first row, the fifth element in the second row, the fourth element in the third and fourth row, and the second element the last row. So, using the size information, we can scan through all of these and identify which is the biggest one in this case it is 43, to be identify the 43 needs to be deleted. And then having deleted 43 then we remove it, and then of course we have to also reduce the size back from 4 to 3. So, we can keep track of the size as in when we manipulate in the previous example also when we had increased 11, we make this 3 into 4. So, again it is an order root N operation.

(Refer Slide Time: 08:34)

## Two dimensional structures

Summary

- `insert()` takes  $O(\sqrt{N})$
- `delete_max()` takes  $O(\sqrt{N})$
- Processing  $N$  jobs takes  $O(N\sqrt{N})$  *prev.  $O(N^2)$*


Can we do better?  $N^{3/2}$

12	17	29	31	40
8	19	22	33	37
10	13	14		
13	20	25	43	
6	11			

So, we have now achieved a data structure, which keeps track of elements in a priority queue where insert takes order root  $N$  time, delete max takes order root  $N$  time, and therefore, now processing a sequence of  $N$  jobs takes  $N$  root  $N$  time. Remember that previously it was order  $N$  square. So, we have reduce from order  $N$  square to order  $N^{3/2}$ , if you want the ((Refer Time: 08:54)). So, this is just a sampler to explain that a two dimensional structure can give you significant savings over a linear search. So, of course we are not going to be happy with this, others you would have just stop with this. So, we can actually do much better than  $N^{3/2}$ , and this is what we are going to discuss in a later lecture.

(Refer Slide Time: 09:14)

## Trees



- Maintain a special kind of binary tree called a **heap**
- **Balanced**:  $N$  node tree has height  $\log N$
- Both `insert()` and `delete_max()` take  $O(\log N)$
- Processing  $N$  jobs takes time  $O(N \log N)$
- Truly flexible, need not fix upper bound for  $N$  in advance

To give you a preview, what we are going to do is to maintain it not in a simple array or a square matrix like this, but in a special kind of binary tree called a heap. So, this will be a binary tree. So, it will have structure like this, and it will be balanced. So, basically all the pass will roughly have the same length, and what this will mean is that the height of the tree will be logarithmic in the size of the tree. And this will make both insert and delete max take  $\log N$  operations, and this will given overall down for  $N$  operations of  $N \log N$ .

The other thing is that we actually maintain it as a dynamic tree like this, we do not have to make an assumption as we did not or simple solution that we just proposed, where we upper bound  $N$ . We can have a solution where the key can grow as large as we want. So, long as we make sure that we grow it in a systematic way to keep it balanced. So, this is what we will look at next, we will look at this data structure called a heap which implements a priority queue by maintaining a set type of binary tree.